COSC 3319-01 Lab 2 Report

Name: Darren William Lehmann

Course & Section: COSC 3319-01

Meeting Days: MWF 8:00-8:50am

Submission Date: October 26, 2025

Grading Option Completed: A+

Contents

1 Option C: Static Allocation

1.1 Objectives

- **Two static arrays: SortByJob[jobTypeCount]int (heads) and SortSpace[maxN+1] (node pool, 1..N, 0=null)**
  All nodes were stored in a fixed-size array, and each job type's head was tracked in SortByJob. This setup made memory usage predictable.

- **Simple allocator Avail from 1 upward. No dynamic allocation in core path**
  The allocator simply incremented an index counter (Avail) to hand out the next available slot, ensuring each node came from the SortSpace[maxN+1].

- **Parse to EOF: Name Job Age. Reject unknown jobs gracefully**
  The loader scanned each line until the EOF, splitting it into name, job, and age fields, and skipped any unrecognized job types to avoid invalid inserts.

- **Maintain circular list per JobType with order (Age asc, Name asc)**
  Each job type had its own circular linked list, and the InsertSorted function placed new nodes in sorted order by age first, and by name alphabetically when ages matched.

- **Print each non-empty bucket in definition order**
  After all records were loaded, PrintBuckets iterated through each job type in order and displayed every circular list that contained employees.

1.2 Program Design

The program is organized as follows:

```
lab2/
 cmd/lab2/main.go           # CLI: parses flags and calls loaders/printers
 internal/types/jobs.go     # JobType enum and parsing helpers
 internal/core/c_static.go  # Option C: static arrays and circular lists
 internal/io/loader.go      # Loaders for each mode
 docs\ContainersCSpring25Mission.txt          # Input file
```

1.3 Algorithm Overview

The InsertSorted function adds each node to the circular list for its job type. It uses the `less` function to keep the nodes in order (age ascending, then name alphabetically if ages match).

InsertSorted handles four main situations:

- Inserting into an empty list: the new node becomes both the head and tail.

- Adding a new head: the new node comes before the current head; update the tail's next pointer.

- Inserting in the middle: locate the correct spot between existing nodes and adjust links.

- Adding to the tail: place the new node just before the head to maintain circular list.

1.4 Implementation

1.4.1 InsertSorted Function

```
// InsertSorted(idx): insert SortSpace[idx] into the circular list for its JobType
func (st *State) InsertSorted(idx int) {
    // TODO: handle four cases (empty, new head, middle, tail)
    job := st.SortSpace[idx].Job
    head := st.SortByJob[job]

    // EMPTY QUEUE - The new node becomes the head and it points to itself
    if head == 0 {
        st.SortByJob[job] = idx
        st.SortSpace[idx].Next = idx
        return
    }
    // intialize tail for NEW HEAD and TAIL cases
    tail := head
    for st.SortSpace[tail].Next != head {
        tail = st.SortSpace[tail].Next
    }

    // NEW HEAD - new node is smaller than the current node
    // new head inserts before the current head then update pointer
    if st.less(idx, head) {
        st.SortSpace[tail].Next = idx
        st.SortSpace[idx].Next = head
        st.SortByJob[job] = idx
        return
    }
    // MIDDLE - new node belongs somewhere between head and tail
    // loop walks through the list to find insertion
    current := head
    for st.SortSpace[current].Next != head {
        following := st.SortSpace[current].Next
        if st.less(idx, following) {
            st.SortSpace[idx].Next = following
            st.SortSpace[current].Next = idx
            return
        }
        current = following
    }
    // TAIL: if largest element node inserts into tail
    // append after the tail
    st.SortSpace[tail].Next = idx
    st.SortSpace[idx].Next = head

}
```

Listing 1: InsertSorted function implementation

1.4.2 PrintBuckets Function

```go
// PrintBuckets - For each job type in definition order, loop its
   circular list and print each node
func (st *State) PrintBuckets() {
    // TODO
    for job := 0; job < jobTypeCount; job++ {
        head := st.SortByJob[job]
        if head == 0 {
            continue
        }

        fmt.Println("JobType", job)
        current := head
        for {
            node := st.SortSpace[current]
            fmt.Println(node.Name, node.Age)
            current = node.Next
            if current == head {
                break
            }
        }
        fmt.Println()
    }
}
```

Listing 2: PrintBuckets function implementation

1.4.3 Supporting Code

```go
// IN C_STATIC.GO
// less(i, j) - true if SortSpace[i] should appear before SortSpace[j]
   within a bucket
func (st *State) less(i, j int) bool {
    // implement: Age asc, then Name asc
    // TODO
    if st.SortSpace[i].Age != st.SortSpace[j].Age {
        return st.SortSpace[i].Age < st.SortSpace[j].Age
    }
    return st.SortSpace[i].Name < st.SortSpace[j].Name
}
// IN JOBS.GO
// takes string and ties it to the enum of JobType
func ParseJobType(s string) JobType {
    switch s {
    case "Accountant":
        return Accountant
    case "Analysist":
        return Analysist
    case "Manager":
        return Manager
    case "Manufacturing":
        return Manufacturing
    case "Programmer":
        return Programmer
    case "Inventory":
        return Inventory
```

```
    case "Sales":
        return Sales
    case "SoftwareEngineer":
        return SoftwareEngineer
    default:
        return -1
    }
}
// IN LOADER.GO
// opens the file and scans each line for name, job, and age then uses C_STATIC functions
func LoadC(st *core.State, path string) error {
    file, err := os.Open(C:\Users\under\DSA\lab2\docs\ContainersCSpring25Mission.txt)
    if err != nil {
        return err
    }
    defer file.Close()
    scan := bufio.NewScanner(file)

    for scan.Scan() {
        line := scan.Text()
        var name, job string
        var age int
        fmt.Sscan(line, &name, &job, &age)
        jobType := types.ParseJobType(job)
        idx := st.Avail
        st.SortSpace[idx] = core.Node{Name: name, Job: jobType, Age: age}
        st.Avail++
        st.InsertSorted(idx)
    }
    return err
}
```

Listing 3: Supporting Functions: less(i, j), ParseJobTypes(), and LoadC()

1.5 Results

```
JobType 0 // Accountant
Sable 26

JobType 1 // Analysist
Betty 23
Betty 23
Kevin 23
Tom 25

JobType 2 // Manager
Bob 43
Ben 57

JobType 4 // Programmer
Teddy 17

JobType 6 // Sales
Sable 32
```

```
Donald 36
Dustin 36
Sable 47
```

Listing 4: Output for Option C

2 Option B: Dynamic Allocation for Homogeneous Records

2.1 Objectives

- **Homogeneous record: (Make, Model, Doors)**
  Each input line was read as a uniform structure representing a vehicle's make, model, and number of doors.
- **Map Make → JobType, Doors → Age, Model → Name**
  The make of the vehicle was used to determine which job type it belongs to, the number of doors acted as the age field, and the model name represented the employee's name in the circular list.
- **Dynamic circular lists per JobType; same ordering as C**
  Instead of using static arrays like Option C, memory was allocated dynamically to build separate circular lists for each job type, keeping the same age-ascending and name-ascending order.
- **Parse to EOF; after ingestion, print a clearly labeled per-bucket list**
  The LoadB() read lines until the end of the file and inserted each record into the correct bucket. Once completed, the list for each type of job was printed with clear headings to show its organized content.

2.2 Program Design

2.2.1 Directory Structure

```
lab2/
 cmd/lab2/main.go           # CLI: parses flags and calls loader & printer
 internal/core/c_static.go  # Option C: static arrays and circular lists
 internal/core/b_dynamic.go  # Option B: Homogeneous, Dynamic Circular Lists
 internal/io/loader.go       # LoadB() reads employees and vehicle lines
 internal/types/jobs.go      # JobType enum and parsing helpers
 docs\ContainersBSpring25Mission.txt  # Input file
```

2.3 Data Structure Overview

In Option B, the program uses dynamic memory allocation rather than a single static array like in Option C. Each record read from the input file creates a new node on the heap instead of using a pre-allocated pool. This approach simplifies insertion into the circular lists because there is no fixed-size limit, but searching or maintaining pointers requires traversing dynamically allocated nodes, which can be slightly more involved compared to the indexed static array approach in Option C.

2.4 Implementation

2.4.1 Loader Function (LoadB)

```
// LoadB - Reads "Make Model Doors" lines from a file until EOF
//   Inserts each record into the appropriate dynamic circular list
func LoadB(path string) ([]core.Head, error) {
    // TODO: read "Make Model Doors" lines to EOF; insert into per-JobType rings
    file, err := os.Open(C:\Users\under\DSA\lab2\docs\ContainersBSpring25Mission.txt)
    if err != nil {
        return nil, err
    }
    defer file.Close()
    scan := bufio.NewScanner(file)
    buckets := make([]core.Head, 5)

    for scan.Scan() {
        tok := scan.Text()
        var make, model string
        var doors int
        fmt.Sscan(tok, &make, &model, &doors)
        jobType := core.MapMakeToJobType(make)
        c := &core.CarNode{Make: make, Model: model, Doors: doors, Job: jobType}

        buckets[jobType].InsertSorted(c)
    }
    return buckets, scan.Err()
}
```

Listing 5: LoadB() in loader.go

2.4.2 InsertSorted Function

This Function uses similar logic to Option C just using dynamic list instead of static array.

```
// InsertSorted - Inserts a new CarNode into the circular list

func (h Head) InsertSorted(nCarNode) {

    // EMPTY QUEUE - The new node becomes the head and it points to itself
    if h.head == nil {
        h.head = n
        n.Next = n
        return

    }
    // NEW HEAD - new node is smaller than the current node
    // new head inserts before the current head then update pointer
    if lessCar(n, h.head) {
        tail := h.head
        for tail.Next != h.head {
            tail = tail.Next
```

```
        }
        n.Next = h.head
        tail.Next = n
        h.head = n
        return
    }

    // MIDDLE - new node belongs somewhere between head and tail
    // loop walks through the list to find insertion
    current := h.head
    for current.Next != h.head {
        if lessCar(n, current.Next) {
            n.Next = current.Next
            current.Next = n
            return
        }
        current = current.Next
    }

   // TAIL: if largest element node inserts into tail
    // append after the tail
    n.Next = h.head
    current.Next = n
}
```

Listing 6: InsertSorted for b_dynamic.go

2.4.3 Printing Functions

- `PrintBucketsB`: ascending and descending per JobType
- Print employee name, age, and vehicle

```
// PrintBucketsB - Loops through each bucket and prints nodes
// Shows Make, Model, and Doors for all vehicles in order

func PrintBucketsB(buckets []Head) {
    // TODO
    for job := 0; job < len(buckets); job++ {
        head := buckets[job].head
        if head == nil {
            continue
        }

        fmt.Println("Manufacturer", job)
        current := head
        for {
            fmt.Println(current.Make, current.Model, current.Doors)
            current = current.Next
            if current == head {
                break
            }
        }
        fmt.Println()
    }
}
```

Listing 7: PrintBucketsB.go

2.4.4 Supporting Functions

- JobType parser (string → enum)

```go
// MapMakeToJobType - Converts a vehicle make to a JobType index

func MapMakeToJobType(m string) int {
    // TODO: define mapping of manufacturer to bucket index
    switch m {
    case "Ford":
        return 0
    case "GMC":
        return 1
    case "Dodge":
        return 2
    case "Chevrolet":
        return 3
    default:
        return 4
    }
}
//  lessCar - Returns true if CarNode a should come before CarNode b
//  Comparison: Doors ascending, then Model ascending
func lessCar(a, b *CarNode) bool {
    // TODO: Doors asc, then Model asc
    if a.Doors != b.Doors {
        return a.Doors < b.Doors
    } else {
        return a.Model < b.Model
    }

}
```

Listing 8: Helper functions and Manufacturer definitions

2.5 Results

- Show sample output for ascending and descending JobTypes
- Include employee details and assigned car

```
Manufacturer 0
Ford Expedition 4
Ford Raptor 4
Ford Raptor 4
Ford Expedition 5

Manufacturer 1
GMC Pickup 2

Manufacturer 2
```

```
Dodge Devil 2
Dodge Ram 2
Dodge Charger 4
Dodge Charger 5

Manufacturer 3
Chevrolet Stingray 2
Chevrolet Camaro 4
```

Listing 9: Option B Results

3 Option A: Heterogeneous via Interfaces (Cars & Planes)

3.1 Objectives

- **Dynamic allocation of employees with multiple vehicle types (Car, Plane)**: Each employee is created on the heap as a separate object. Vehicles are assigned dynamically and stored as interface types so that Cars and Planes can coexist in the same slice.
- **Maintain per-JobType circular lists**: Each JobType has its own circular linked list. Employees are inserted into the appropriate list to preserve ordering.
- **Enforce maximum vehicles per employee (capVehicles)**: The program ensures that no employee receives more vehicles than the specified cap. Vehicles beyond the limit are ignored to maintain the constraint.
- **Insert employees in Age ascending, Name ascending order**: The InsertSorted method compares age first and then name to place each employee in the correct position in the circular list.
- **Two-pass printing: ascending and descending JobType**: After all employees are inserted, the program prints the lists first in ascending order of JobType, then in descending order.
- **Handle multiple types of vehicles (Car, Plane)**: Vehicles are represented with a Vehicle interface. The program determines the type and instantiates the correct struct (Car or Plane).
- **Reject unknown jobs or invalid vehicles gracefully**: Any employee or vehicle that does not match the expected types is skipped during loading, preventing errors and keeping the lists clean.

3.2 Program Design

3.2.1 Directory Structure

```
lab2/
 cmd/lab2/main.go           # CLI: parses flags and calls loader & printer
 internal/core/c_static.go  # Option C: static arrays and circular lists
```

```
internal/core/b_dynamic.go  # Option B: Homogeneous, Dynamic Circular Lists
internal/core/a_dynamic.go# Option A: Emp, EmpNode, EmpHead, InsertSorted, Printfunc
internal/io/loader.go       # LoadA() reads employees and vehicle lines
internal/types/jobs.go      # JobType enum and parsing helpers
docs\ContainersASpring25Mission.txt  # Input file
```

3.2.2 Data Structures

- `Vehicle` interface — base abstraction implemented by all vehicle types.
- `Car`, `Plane` structs — concrete types implementing `Vehicle`.
- `Emp` struct — contains `Name, Job, Age, []Vehicle`.
- `EmpNode`, `EmpHead` — circular linked list structures for per-job organization.

3.3 Implementation

3.3.1 Loader Function (LoadA)

- Open input file
- Scan each employee line
- Scan following vehicle line(s)
- Create employee node dynamically
- Append valid vehicles up to cap
- Insert node into correct JobType list

```
1  func LoadA(path string, capVehicles int) ([]core.EmpHead, error) {
2  // The function enforces capVehicles = 1, builds four JobType buckets
3  // (Analysist, Manager, Accountant, Sales), and inserts each employee
4  // in sorted order by Age then Name within the corresponding bucket.
5  // TODO: read employee line then its vehicle lines; enforce capVehicles
       =1
6      buckets := make([]core.EmpHead, 4) // 4 JobTypes
7      file, err := os.Open(C:\Users\under\DSA\lab2\docs\
           ContainersASpring25Mission.txt)
8      if err != nil {
9          return nil, err
10     }
11     defer file.Close()
12
13     scanner := bufio.NewScanner(file)
14     for scanner.Scan() {
15     for scanner.Scan() {
16         // Read employee line
17         fields := strings.Fields(scanner.Text())
18         name := fields[0]
19         job := types.ParseJobType(fields[1])
20         // string to int for age
21         age, _ := strconv.Atoi(fields[2])
```

```
22
23          // Create employee node
24          emp := &core.Emp{
25              Name: name,
26              Job:  int(job),
27              Age:  age,
28              V:    make([]core.Vehicle, 0, capVehicles),
29          }
30
31          // Read next line for vehicle
32          if scanner.Scan() {
33              vfields := strings.Fields(scanner.Text())
34              manu := vfields[0]
35              model := vfields[1]
36              // string to int for doors
37              num, _ := strconv.Atoi(vfields[2])
38              color := vfields[3]
39
40              // Create vehicle (Car or Plane)
41              var vehicle core.Vehicle
42              if core.IsPlaneManufacturer(manu) {
43                  vehicle = &core.Plane{Manufacturer: manu, Model: model,
                        Engines: num, Color: color}
44              } else {
45                  vehicle = &core.Car{Manufacturer: manu, Model: model,
                        Doors: num, Color: color}
46              }
47
48              // Append vehicle to employee
49              if len(emp.V) < capVehicles {
50                  emp.V = append(emp.V, vehicle)
51              }
52          }
53
54          // Insert employee into the correct bucket
55          buckets[emp.Job].InsertSorted(emp)
56      }
57
58      return buckets, scanner.Err()
59  }
```

Listing 10: Loader for Option A

3.3.2 InsertSorted Function

- Handles empty list
- Insert at head
- Insert in middle
- Insert at tail
- Maintains Age ascending, Name ascending order

```go
func (h *EmpHead) InsertSorted(e *Emp) {
// InsertSorted - inserts an employee into a circular linked list
// Handles four insertion cases: empty list, head, middle, and tail.
// TODO: four insertion cases in circular ring
// EMPTY LIST
    if h.head == nil {
        eNode := &EmpNode{E: e}
        eNode.next = eNode // points to itself
        h.head = eNode
        return
    }
    prev := h.head
    curr := h.head.next
    // MIDDLE BETWEEN TWO NODES also performs NEW HEAD in non-empty
       list
    for {
        if lessEmp(e, curr.E) {
            // Insert between prev and curr middle
            newNode := &EmpNode{E: e, next: curr}
            prev.next = newNode
            if curr == h.head && !lessEmp(e, h.head.E) {
                h.head = newNode
            }
            return
        }
        prev = curr
        curr = curr.next
        if curr == h.head.next {
            break // back to start
        }
    }

    // TAIL insert if not yet inserted
    newNode := &EmpNode{E: e, next: h.head.next}
    prev.next = newNode
}
```

Listing 11: InsertSorted for a_dynamic.go

3.3.3 Printing Functions

- Prints name, age, vehicle type, vehicle model, vehicle doors, and vehicle color

```go

func PrintAscending(buckets []EmpHead) {
    for i := 0; i < len(buckets); i++ {
        fmt.Println("JobType", i)
        h := buckets[i]
        if h.Head != nil {
            curr := h.Head
            for {
                fmt.Println("␣␣", curr.E.Name, curr.E.Age)
                for _, v := range curr.E.V {
                    switch veh := v.(type) {
                    case *Car:
```

```
                        fmt.Printf("␣␣␣␣␣␣Car:␣%s␣%s␣Doors:%d␣Color:%s\
                            n",
                            veh.Manufacturer, veh.Model, veh.Doors, veh
                                .Color)
                    case *Plane:
                        fmt.Printf("␣␣␣␣␣␣Plane:␣%s␣%s␣Engines:%d␣Color
                            :%s\n",
                            veh.Manufacturer, veh.Model, veh.Engines,
                                veh.Color)
                    case *Motorcycle:
                        fmt.Printf("␣␣␣␣␣␣Motorcycle:␣%s␣%s␣Wheels:%d␣
                            Color:%s\n",
                            veh.Manufacturer, veh.Model, veh.Wheels,
                                veh.Color)
                    }
                }
                curr = curr.Next // Move to next employee AFTER
                    printing all vehicles
                if curr == h.Head {
                    break
                }
            }
        }
    }
}

func PrintDescending(buckets []EmpHead) {
    for i := len(buckets) - 1; i >= 0; i-- {
        h := buckets[i]
        fmt.Println("JobType", i)
        if h.Head != nil {
            curr := h.Head
            for {
                fmt.Println("␣␣", curr.E.Name, curr.E.Age)
                for _, v := range curr.E.V {
                    switch veh := v.(type) {
                    case *Car:
                        fmt.Printf("␣␣␣␣␣␣Car:␣%s␣%s␣Doors:%d␣Color:%s\
                            n",
                            veh.Manufacturer, veh.Model, veh.Doors, veh
                                .Color)
                    case *Plane:
                        fmt.Printf("␣␣␣␣␣␣Plane:␣%s␣%s␣Engines:%d␣Color
                            :%s\n",
                            veh.Manufacturer, veh.Model, veh.Engines,
                                veh.Color)
                    case *Motorcycle:
                        fmt.Printf("␣␣␣␣␣␣Motorcycle:␣%s␣%s␣Wheels:%d␣
                            Color:%s\n",
                            veh.Manufacturer, veh.Model, veh.Wheels,
                                veh.Color)
                    }
                }
                curr = curr.Next // Move to next employee AFTER
                    printing all vehicles
                if curr == h.Head {
                    break
                }
```

```
            }
        }
    }
}
```

Listing 12: PrintAsc and PrintDesc

3.3.4 Supporting Functions

```
//  in a_dynamic.go
// TODO: Age asc then Name asc if ages equal

func lessEmp(x, y *Emp) bool {
    if x.Age != y.Age {
        return x.Age < y.Age
    }
    return x.Name < y.Name
}
// helps for finding plane manufacturer in Loader
func IsPlaneManufacturer(m string) bool {
    // TODO: whitelist (e.g., GeneralDynamics, Lockheed, Boeing,
       Grumman)
    whitelist := map[string]bool{
        "GeneralDynamics": true,
        "Lockheed":        true,
        "Boeing":          true,
        "Grumman":         true,
    }
    return whitelist[m]
}
```

Listing 13: Option A Support Functions lessEmp() and IsPlaneManufacturer()

3.4 Results

```
// results_A.txt
ASCENDING ORDER
JobType 0
   Sable 26
      Car: GMC Pickup Doors:2 Color:White
JobType 1
   Kevin 23
      Car: Ford Expedition Doors:5 Color:Blue
   Tom 23
      Car: Dodge Charger Doors:4 Color:Black
   Betty 23
      Car: Chevrolet Stingray Doors:2 Color:Red
JobType 2
   Ben 57
      Car: Ford Raptor Doors:4 Color:Red
   Bob 43
      Car: Dodge Devil Doors:2 Color:Orange
   Bob 44
      Car: Chevrolet Stingray Doors:2 Color:Blue
JobType 3
```

```
   Donald 36
      Car: Ford Expedition Doors:4 Color:White
   Sable 47
      Car: Dodge Ram Doors:5 Color:Black
   Sable 32
      Car: Dodge Ram Doors:2 Color:White
   Dustin 36
      Car: Ford Raptor Doors:4 Color:Silver
JobType 4
   Teddy 17
      Car: Chevrolet Camaro Doors:4 Color:Black

DESCENDING ORDER
JobType 4
   Teddy 17
      Car: Chevrolet Camaro Doors:4 Color:Black
JobType 3
   Donald 36
      Car: Ford Expedition Doors:4 Color:White
   Sable 47
      Car: Dodge Ram Doors:5 Color:Black
   Sable 32
      Car: Dodge Ram Doors:2 Color:White
   Dustin 36
      Car: Ford Raptor Doors:4 Color:Silver
JobType 2
   Ben 57
      Car: Ford Raptor Doors:4 Color:Red
   Bob 43
      Car: Dodge Devil Doors:2 Color:Orange
   Bob 44
      Car: Chevrolet Stingray Doors:2 Color:Blue
JobType 1
   Kevin 23
      Car: Ford Expedition Doors:5 Color:Blue
   Tom 23
      Car: Dodge Charger Doors:4 Color:Black
   Betty 23
      Car: Chevrolet Stingray Doors:2 Color:Red
JobType 0
   Sable 26
      Car: GMC Pickup Doors:2 Color:White
```

Listing 14: Option A Results

4 Option A+: Extension of Option A

4.1 Objectives

- **Allow 2 vehicles/employee (e.g., cap at 2)**
  Each employee is allocated a slice of vehicles with a maximum capacity, ensuring no more than two vehicles can be assigned. The loader function checks this limit before appending new vehicles.

- **Add one extra type (e.g., Motorcycle) with a simple mapping rule**

Vehicles are classified into Car, Plane, or Motorcycle based on the manufacturer or type keyword. Motorcycles are assigned to existing employees without creating a new job type, using the same insertion logic.

- **Same two-pass printing as A**
  Employees are printed by JobType in ascending order first, then descending order. For each employee, all assigned vehicles are printed with full attributes.

4.2 Program Design

4.2.1 Directory Structure

```
lab2/
 cmd/lab2/main.go           # CLI: parses flags and calls loader & printer
 internal/core/c_static.go  # Option C: static arrays and circular lists
  internal/core/b_dynamic.go  # Option B: Homogeneous, Dynamic Circular Lists
 internal/core/a_dynamic.go  # Option A: Emp, EmpNode, EmpHead, InsertSorted, Print
 internal/io/loader.go       # LoadA() reads employees and vehicle lines
 internal/types/jobs.go      # JobType enum and parsing helpers
 docs\ContainersASpring25Mission.txt  # Input file
```

4.2.2 Data Structures

- `EmpHead`: circular linked list head per JobType.
- `EmpNode`: contains an `Emp` record and pointer to the next node.
- `Emp`: stores employee name, job, age, and up to two vehicles.
- `Vehicle` interface: defines common behavior for vehicle types.
- `Car`, `Plane`, `Motorcycle`: concrete implementations of `Vehicle`.

4.3 Implementation

4.3.1 Data Structures Added

```
1     type Motorcycle struct {
2     Manufacturer, Model, Color string
3     Wheels                     int
4 }
```

Listing 15: Data Structure implementation for Motorcycles

4.3.2 Loader Function (LoadA)

- Allocates employees and their associated vehicles.
- Detect manufacturer type (Car, Plane, or Motorcycle).
- Attach vehicle to the last processed employee if under the 2-vehicle limit.

```go
func LoadA(path string, capVehicles int) ([]core.EmpHead, error) {
    buckets := make([]core.EmpHead, 5) // 5 JobTypes
    file, err := os.Open(`C:\Users\will\Desktop\coding\lab2\docs\
       ContainersA+Spring25Mission.txt`)
    if err != nil {
        return nil, err
    }
    defer file.Close()

    scanner := bufio.NewScanner(file)
    var currentEmp *core.Emp

    for scanner.Scan() {
        fields := strings.Fields(scanner.Text())

        switch len(fields) {
        case 3: // employee line
            name := fields[0]
            job := types.ParseJobType(fields[1])
            // string to int for age
            age, _ := strconv.Atoi(fields[2])

            if job < 0 || int(job) >= len(buckets) {
                continue // skip unknown jobs
            }

            h := &buckets[job]
            found := false
            if h.Head != nil {
                curr := h.Head
                for {
                    if curr.E.Name == name && curr.E.Age == age {
                        currentEmp = curr.E
                        found = true
                        break
                    }
                    curr = curr.Next
                    if curr == h.Head {
                        break
                    }
                }
            }

            if !found {
                currentEmp = &core.Emp{
                    Name: name,
                    Job:  int(job),
                    Age:  age,
                    V:    make([]core.Vehicle, 0, capVehicles),
                }
                h.InsertSorted(currentEmp)
            }

        case 4: // vehicle line
            if currentEmp == nil || len(currentEmp.V) >= capVehicles {
                continue
            }
```

```
            manu := fields[0]
            model := fields[1]
            // string to int for age
            num, _ := strconv.Atoi(fields[2])
            color := fields[3]

            var vehicle core.Vehicle
            switch {
            case core.IsPlaneManufacturer(manu):
                vehicle = &core.Plane{Manufacturer: manu, Model: model,
                    Color: color, Engines: num}
            case core.IsMotorcycleManufacturer(manu):
                vehicle = &core.Motorcycle{Manufacturer: manu, Model:
                   model, Color: color, Wheels: num}
            default:
                vehicle = &core.Car{Manufacturer: manu, Model: model,
                   Color: color, Doors: num}
            }

            currentEmp.V = append(currentEmp.V, vehicle)
        }
    }

    return buckets, scanner.Err()
}
```

Listing 16: LoadA function implementation for Option A+

4.3.3 Supporting Functions

- `Describe`: Gives manufacture and model of Motorcyle together
- `Kind`: Return type of vehicle as string "Motorcyle"
- `IsMotorcycleManufacturer`: identifies motorcycle manufacturers.

```

// new functions for motorcycles
func (m *Motorcycle) Kind() string     { return "Motorcycle" }

func (m *Motorcycle) Describe() string { return m.Manufacturer + "␣" +
   m.Model }

func IsMotorcycleManufacturer(m string) bool {
    whitelist := map[string]bool{
        "Harley":   true,
        "Yamaha":   true,
        "Kawasaki": true,
    }
    return whitelist[m]
}
```

Listing 17: Supporting functions for Option A+

4.3.4 A+ Input File

```
// added motorcycles and changed the order of employees, cars, and
   planes
Kevin Analysist 23
Ford Expedition 5 Blue
Yamaha R1 2 Red
Ben Manager 57
Ford Raptor 4 Red
Sable Accountant 26
GMC Pickup 2 White
Honda CBR500R 2 Blue
Bob Manager 43
Dodge Devil 2 Orange
Harley Davidson Street750 2 Black
Teddy Programmer 17
Chevrolet Camaro 4 Black
Donald Sales 36
Ford Expedition 4 White
Dustin Sales 36
Ford Raptor 4 Silver
Suzuki GSX 2 Yellow
Betty Analysist 23
Chevrolet Stingray 2 Red
Tom Analysist 23
Dodge Charger 4 Black
Betty Analysist 23
Bob Manager 44
Chevrolet Stingray 2 Blue
Sable Sales 32
Dodge Ram 2 White
Sable Sales 47
Dodge Ram 5 Black
GeneralDynamics F-16 1 Silver
GeneralDynamics F-16 1 Camo
Grumman Commercial 4 White
Lockheed F35 1 Silver
Boeing 747 5 Silver
```

Listing 18: Input file for Option A+

4.4 Results

- Verified multiple vehicles per employee (limit of 2) across various JobTypes.
- Confirmed correct circular traversal in both ascending and descending order.
- Output shows Cars, Planes, and Motorcycles linked to employees.

```
ASCENDING ORDER
JobType 0
   Sable 26
      Car: GMC Pickup Doors:2 Color:White
      Car: Honda CBR500R Doors:2 Color:Blue
JobType 1
   Kevin 23
```

```
      Car: Ford Expedition Doors:5 Color:Blue
      Motorcycle: Yamaha R1 Wheels:2 Color:Red
   Tom 23
      Car: Dodge Charger Doors:4 Color:Black
   Betty 23
      Car: Chevrolet Stingray Doors:2 Color:Red
JobType 2
   Ben 57
      Car: Ford Raptor Doors:4 Color:Red
   Bob 43
      Car: Dodge Devil Doors:2 Color:Orange
   Bob 44
      Car: Chevrolet Stingray Doors:2 Color:Blue
JobType 3
   Donald 36
      Car: Ford Expedition Doors:4 Color:White
   Sable 47
      Car: Dodge Ram Doors:5 Color:Black
      Plane: GeneralDynamics F-16 Engines:1 Color:Silver
   Sable 32
      Car: Dodge Ram Doors:2 Color:White
   Dustin 36
      Car: Ford Raptor Doors:4 Color:Silver
      Car: Suzuki GSX Doors:2 Color:Yellow
JobType 4
   Teddy 17
      Car: Chevrolet Camaro Doors:4 Color:Black

DESCENDING ORDER
JobType 4
   Teddy 17
      Car: Chevrolet Camaro Doors:4 Color:Black
JobType 3
   Donald 36
      Car: Ford Expedition Doors:4 Color:White
   Sable 47
      Car: Dodge Ram Doors:5 Color:Black
      Plane: GeneralDynamics F-16 Engines:1 Color:Silver
   Sable 32
      Car: Dodge Ram Doors:2 Color:White
   Dustin 36
      Car: Ford Raptor Doors:4 Color:Silver
      Car: Suzuki GSX Doors:2 Color:Yellow
JobType 2
   Ben 57
      Car: Ford Raptor Doors:4 Color:Red
   Bob 43
      Car: Dodge Devil Doors:2 Color:Orange
   Bob 44
      Car: Chevrolet Stingray Doors:2 Color:Blue
JobType 1
   Kevin 23
      Car: Ford Expedition Doors:5 Color:Blue
      Motorcycle: Yamaha R1 Wheels:2 Color:Red
   Tom 23
      Car: Dodge Charger Doors:4 Color:Black
   Betty 23
      Car: Chevrolet Stingray Doors:2 Color:Red
```

```
JobType 0
   Sable 26
      Car: GMC Pickup Doors:2 Color:White
      Car: Honda CBR500R Doors:2 Color:Blue
```

Listing 19: Sample Output for Option A+